AI in the Sciences and Engineering

Introduction to Hybrid Workflows – Part 2

Spring Semester 2024

Siddhartha Mishra
Ben Moseley
ETH zürich

Recap – autodifferentiation

```python
def Hybrid_NS_solver(u_0, p_0, rho, nu, theta):
    "Pseudocode for solving NS equation, with NN correction"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

        u_t, p_t = (u_t, p_t) + NN(u_t, p_t, theta)

    return u_t, p_t

theta.requires_grad_(True)
u_T,_ = Hybrid_NS_solver(u_0, p_0, rho, nu, theta)
loss = loss_fn(u_T, u_T_true)
dtheta = torch.autograd.grad(loss, theta)
# for learning theta (training NN)
```

Many (scientific) programs can be thought of as vector functions composed of many primitive operations:

$$\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{f}_N \circ, \dots, \circ\, \boldsymbol{f}_2 \circ \boldsymbol{f}_1(\boldsymbol{x})$$

Recap – autodifferentiation

```python
def Hybrid_NS_solver(u_0, p_0, rho, nu, theta):
    "Pseudocode for solving NS equation, with NN correction"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

        u_t, p_t = (u_t, p_t) + NN(u_t, p_t, theta)

    return u_t, p_t

theta.requires_grad_(True)
u_T,_ = Hybrid_NS_solver(u_0, p_0, rho, nu, theta)
loss = loss_fn(u_T, u_T_true)
dtheta = torch.autograd.grad(loss, theta)
# for learning theta (training NN)
```

Many (scientific) programs can be thought of as vector functions composed of many primitive operations:

$$\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{f}_N \circ, \dots, \circ \boldsymbol{f}_2 \circ \boldsymbol{f}_1(\boldsymbol{x})$$

Autodifferentiation allows us to efficiently compute:

- The vector-Jacobian product (vjp)

$$\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$$

- The Jacobian-vector product (jvp)

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \boldsymbol{v}$$

Recap – autodifferentiation

```python
def Hybrid_NS_solver(u_0, p_0, rho, nu, theta):
    "Pseudocode for solving NS equation, with NN correction"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

        u_t, p_t = (u_t, p_t) + NN(u_t, p_t, theta)

    return u_t, p_t

theta.requires_grad_(True)
u_T,_ = Hybrid_NS_solver(u_0, p_0, rho, nu, theta)
loss = loss_fn(u_T, u_T_true)
dtheta = torch.autograd.grad(loss, theta)
# for learning theta (training NN)
```

Computes vector-Jacobian product, $1\frac{\partial L}{\partial \boldsymbol{\theta}}$

Many (scientific) programs can be thought of as vector functions composed of many primitive operations:

$$\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{f}_N \circ, \dots, \circ \boldsymbol{f}_2 \circ \boldsymbol{f}_1(\boldsymbol{x})$$

Autodifferentiation allows us to efficiently compute:

- The vector-Jacobian product (vjp)

$$\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$$

- The Jacobian-vector product (jvp)

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \boldsymbol{v}$$

Recap - vector-Jacobian product

vjp:

$$\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \boldsymbol{v}^T \frac{\partial \boldsymbol{f}_N}{\partial \boldsymbol{f}_{N-1}}, \dots, \frac{\partial \boldsymbol{f}_2}{\partial \boldsymbol{f}_1} \frac{\partial \boldsymbol{f}_1}{\partial \boldsymbol{x}}$$

We can compute $\boldsymbol{v}^T \frac{\partial y}{\partial x}$ by iteratively computing vector-Jacobian products, from left to right (reverse-mode):

Starting with $\boldsymbol{v}^T$,

$$\boldsymbol{v}^T \leftarrow \boldsymbol{v}^T \frac{\partial \boldsymbol{f}_N}{\partial \boldsymbol{f}_{N-1}}$$

$$\boldsymbol{v}^T \leftarrow \boldsymbol{v}^T \frac{\partial \boldsymbol{f}_{N-1}}{\partial \boldsymbol{f}_{N-2}}$$

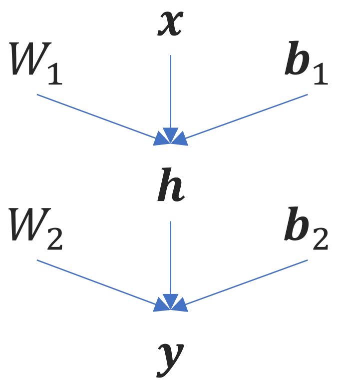
$$\dots$$

$$\boldsymbol{v}^T \leftarrow \boldsymbol{v}^T \frac{\partial \boldsymbol{f}_1}{\partial \boldsymbol{x}}$$

- We only need to define the **vjp** for each **primitive operation** to compute $\boldsymbol{v}^T \frac{\partial y}{\partial x}$
- Usually, we do not need to explicitly compute the full intermediate Jacobians $\frac{\partial f_i}{\partial f_{i-1}}$

Autodiff in practice

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$



1) Decompose given function into its **primitive** operations

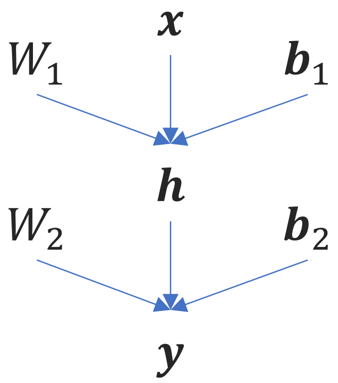
2) Build a **directed graph** of these operations

PyTorch

TensorFlow

Autodiff in practice

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$



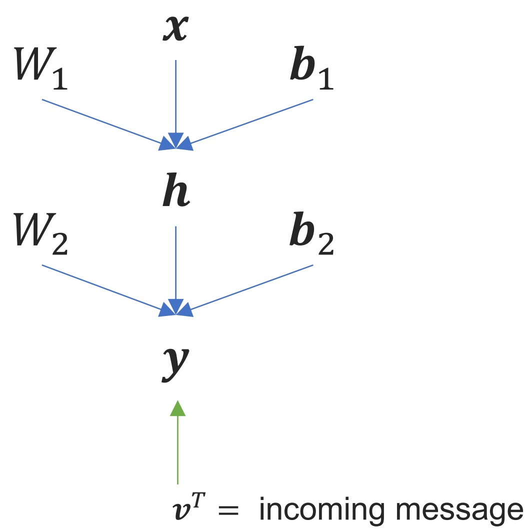
1) Decompose given function into its **primitive** operations

2) Build a **directed graph** of these operations

3) For each primitive operation, define
    1) Forward operation
    2) vector-Jacobian product
    3) Jacobian-vector product

Autodiff in practice

$$\boldsymbol{y} = W_2 \sigma(W_1 \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$



1) Decompose given function into its **primitive** operations

2) Build a **directed graph** of these operations

3) For each primitive operation, define
    1) Forward operation
    2) vector-Jacobian product
    3) Jacobian-vector product

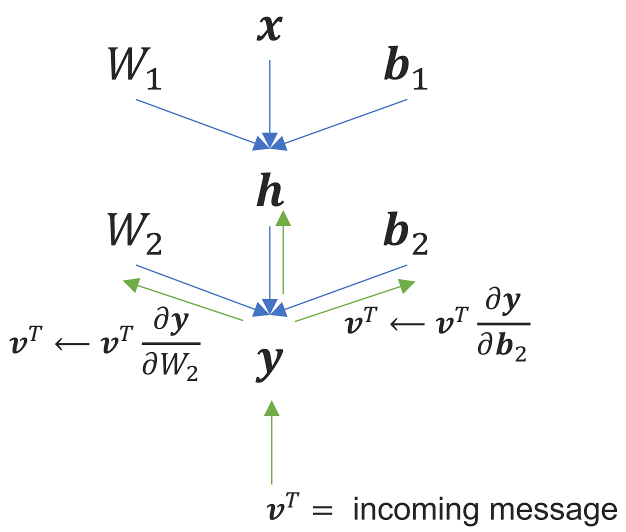
4) Evaluate the vjp or jvp of the function by applying the **chain rule (=message passing)** through the graph
    1) Forwards for jvp
    2) Backwards for vjp

PyTorch

TensorFlow

Autodiff in practice

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$



PyTorch

TensorFlow

1) Decompose given function into its **primitive** operations

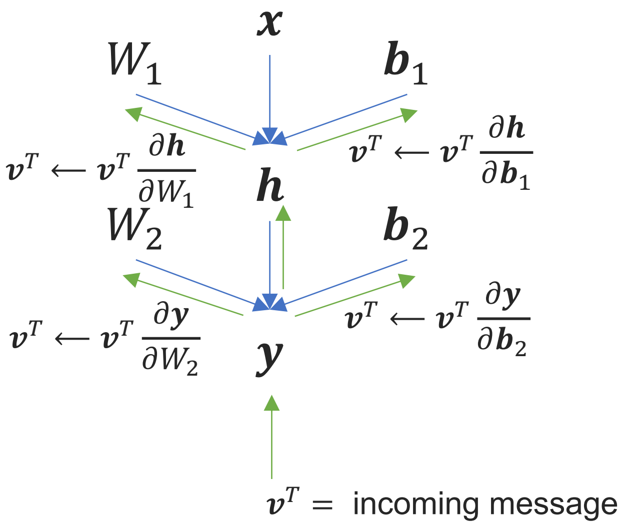
2) Build a **directed graph** of these operations

3) For each primitive operation, define
   1) Forward operation
   2) vector-Jacobian product
   3) Jacobian-vector product

4) Evaluate the vjp or jvp of the function by applying the **chain rule (=message passing)** through the graph
   1) Forwards for jvp
   2) Backwards for vjp

Autodiff in practice

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$



1) Decompose given function into its **primitive** operations

2) Build a **directed graph** of these operations

3) For each primitive operation, define
   1) Forward operation
   2) vector-Jacobian product
   3) Jacobian-vector product

4) Evaluate the vjp or jvp of the function by applying the **chain rule (=message passing)** through the graph
   1) Forwards for jvp
   2) Backwards for vjp

Autodiff in practice

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$

$\boldsymbol{v} =$ incoming message

1) Decompose given function into its **primitive** operations

2) Build a **directed graph** of these operations

3) For each primitive operation, define
   1) Forward operation
   2) vector-Jacobian product
   3) Jacobian-vector product

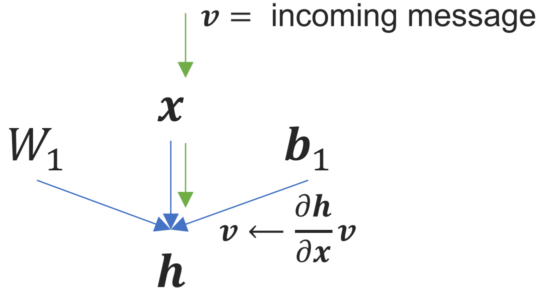
4) Evaluate the vjp or jvp of the function by applying the **chain rule (=message passing)** through the graph
   1) Forwards for jvp
   2) Backwards for vjp

PyTorch

TensorFlow

Autodiff in practice

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$



1) Decompose given function into its **primitive** operations

2) Build a **directed graph** of these operations

3) For each primitive operation, define
   1) Forward operation
   2) vector-Jacobian product
   3) Jacobian-vector product

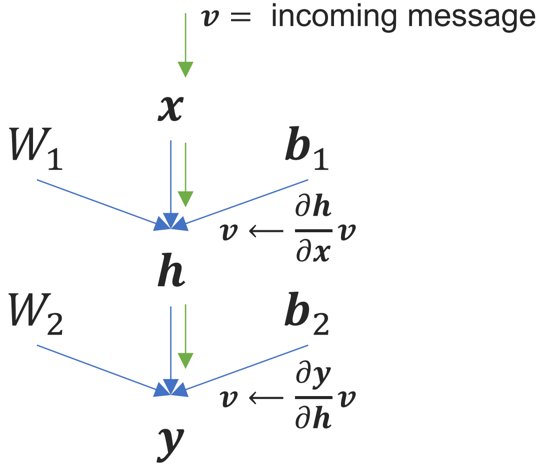
4) Evaluate the vjp or jvp of the function by applying the **chain rule (=message passing)** through the graph
   1) Forwards for jvp
   2) Backwards for vjp

PyTorch

TensorFlow

Autodiff in practice

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$



PyTorch

TensorFlow

1) Decompose given function into its **primitive** operations

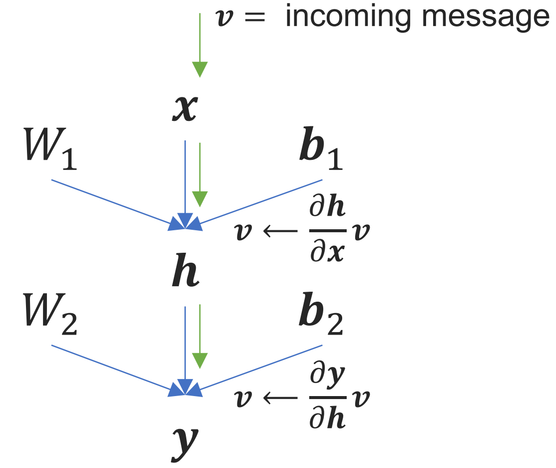
2) Build a **directed graph** of these operations

3) For each primitive operation, define
   1) Forward operation
   2) vector-Jacobian product
   3) Jacobian-vector product

4) Evaluate the vjp or jvp of the function by applying the **chain rule (=message passing)** through the graph
   1) Forwards for jvp
   2) Backwards for vjp

Autodiff in practice

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$



1) Decompose given function into its **primitive** operations

2) Build a **directed graph** of these operations

3) For each primitive operation, define
   1) Forward operation
   2) vector-Jacobian product
   3) Jacobian-vector product

4) Evaluate the vjp or jvp of the function by applying the **chain rule (=message passing)** through the graph
   1) Forwards for jvp
   2) Backwards for vjp

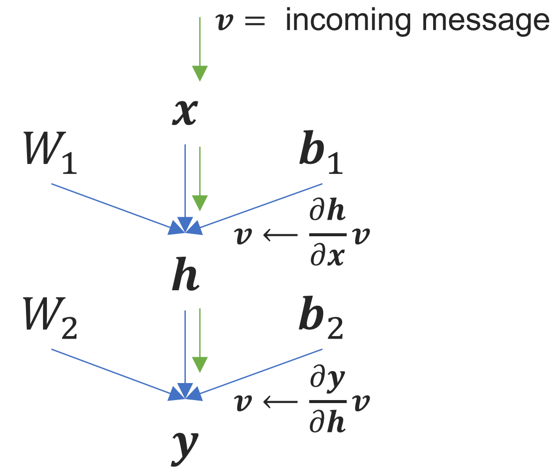
PyTorch

TensorFlow

- How does required memory scale with depth of forward computation for vjp vs jvp?

Autodiff in practice

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$



1) Decompose given function into its **primitive** operations

2) Build a **directed graph** of these operations

3) For each primitive operation, define
   1) Forward operation
   2) vector-Jacobian product
   3) Jacobian-vector product

4) Evaluate the vjp or jvp of the function by applying the **chain rule (=message passing)** through the graph
   1) Forwards for jvp
   2) Backwards for vjp

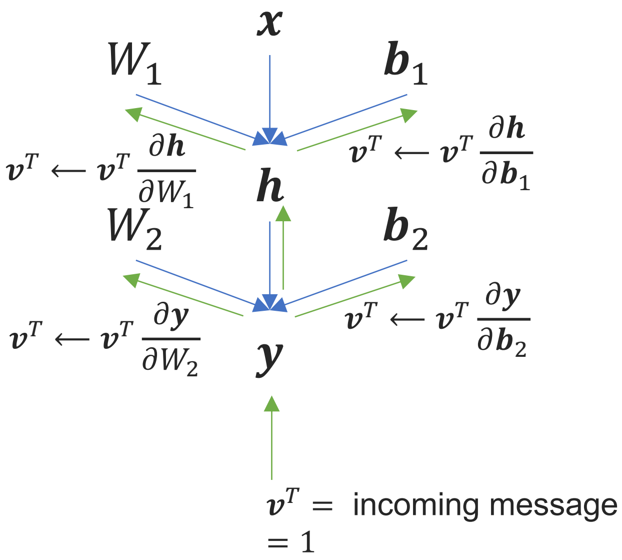
PyTorch

TensorFlow

- How does required memory scale with depth of forward computation for vjp vs jvp?
- vjp: memory scales linearly with depth (need to store forward computations)
- jvp: memory independent of depth (can compute jvp alongside forward pass)

Autodiff in practice

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$



loss.backward()

PyTorch

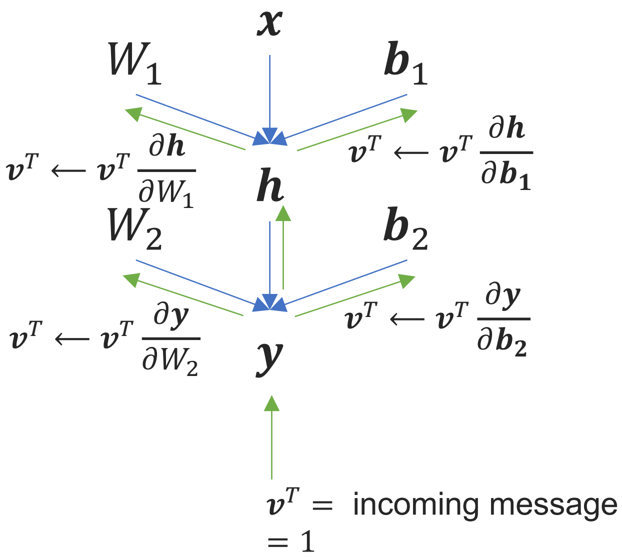
torch.autograd.grad(*outputs*, *inputs*, *grad_outputs=None*, *retain_graph=None*, *create_graph=False*, *only_inputs=True*, *allow_unused=None*, *is_grads_batched=False*, *materialize_grads=False*) [SOURCE]

Computes and returns the sum of gradients of outputs with respect to the inputs.

`grad_outputs` should be a sequence of length matching `output` containing the “vector” in vector-Jacobian product, usually the pre-computed gradients w.r.t. each of the outputs. If an output doesn’t require_grad, then the gradient can be `None`).

Autodiff in practice

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$



loss.backward()

PyTorch

torch.autograd.grad(*outputs*, *inputs*, *grad_outputs=None*, *retain_graph=None*, *create_graph=False*, *only_inputs=True*, *allow_unused=None*, *is_grads_batched=False*, *materialize_grads=False*) [SOURCE]

Computes and returns the sum of gradients of outputs with respect to the inputs.

`grad_outputs` should be a sequence of length matching `output` containing the "vector" in vector-Jacobian product, usually the pre-computed gradients w.r.t. each of the outputs. If an output doesn't require_grad, then the gradient can be `None`).

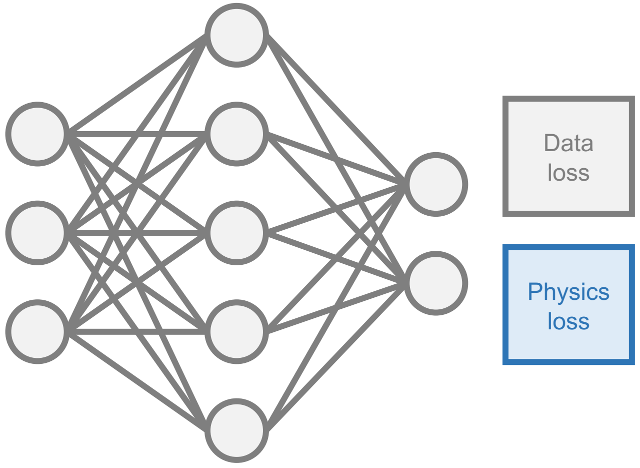
Note autodiff is **not**

- Symbolic differentiation
- Finite differences

It is a way of efficiently computing exact gradients!

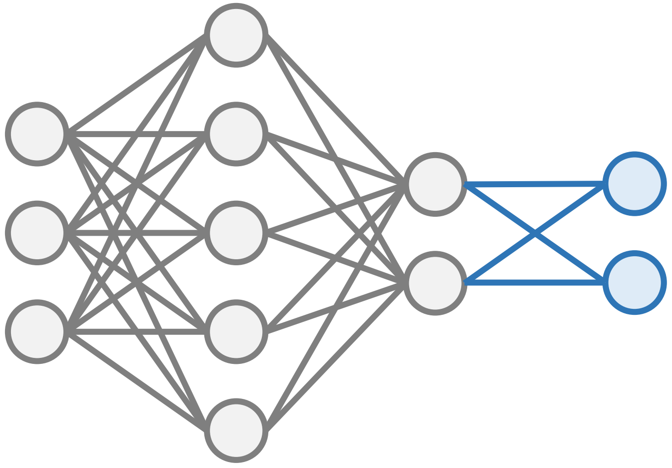
Recap – ways to incorporate scientific principles into machine learning

Loss function



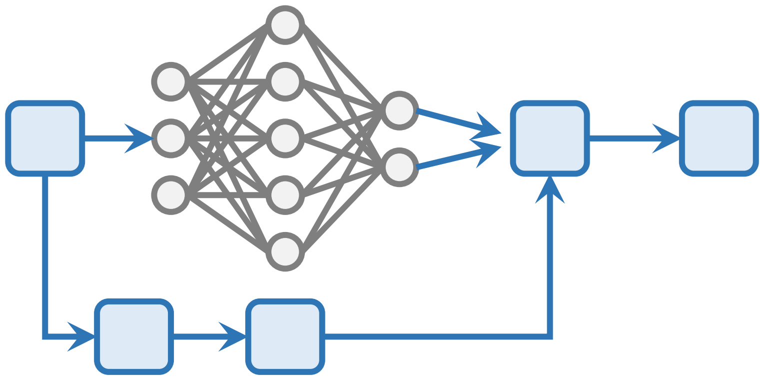
Example:
Physics-informed neural networks
(add governing equations to loss function)

Architecture



Example:
Encoding symmetries / conservation laws
(e.g. energy conservation, rotational invariance), operator learning

Hybrid approaches



Example:
Neural differential equations
(incorporating neural networks into PDE models)

Recap – hybrid approaches

Advantages of DNNs

- Usually very **fast** (once trained)
- Can represent highly **non-linear** functions

Limitations of DNNs

- Often lots of **training data required**
- Can be hard to **optimise**
- Can be hard to **interpret**
- Often struggle to **generalise**

General advice

Use DNNs to:

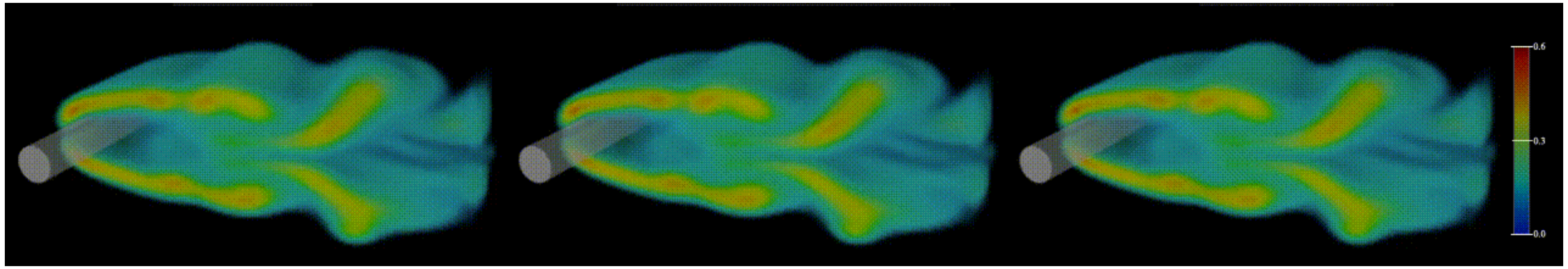
1) **Accelerate** your workflow, or
2) Learn the **parts** you are unsure of / have incomplete knowledge

Entirely replacing your existing workflow with a DNN may **not** be a good idea!

Key idea: **directly incorporate** DNNs into a traditional algorithm **= hybrid approach**

Recap – hybrid Navier-Stokes solver

Low fidelity FD solver | Hybrid approach | High fidelity FD solver



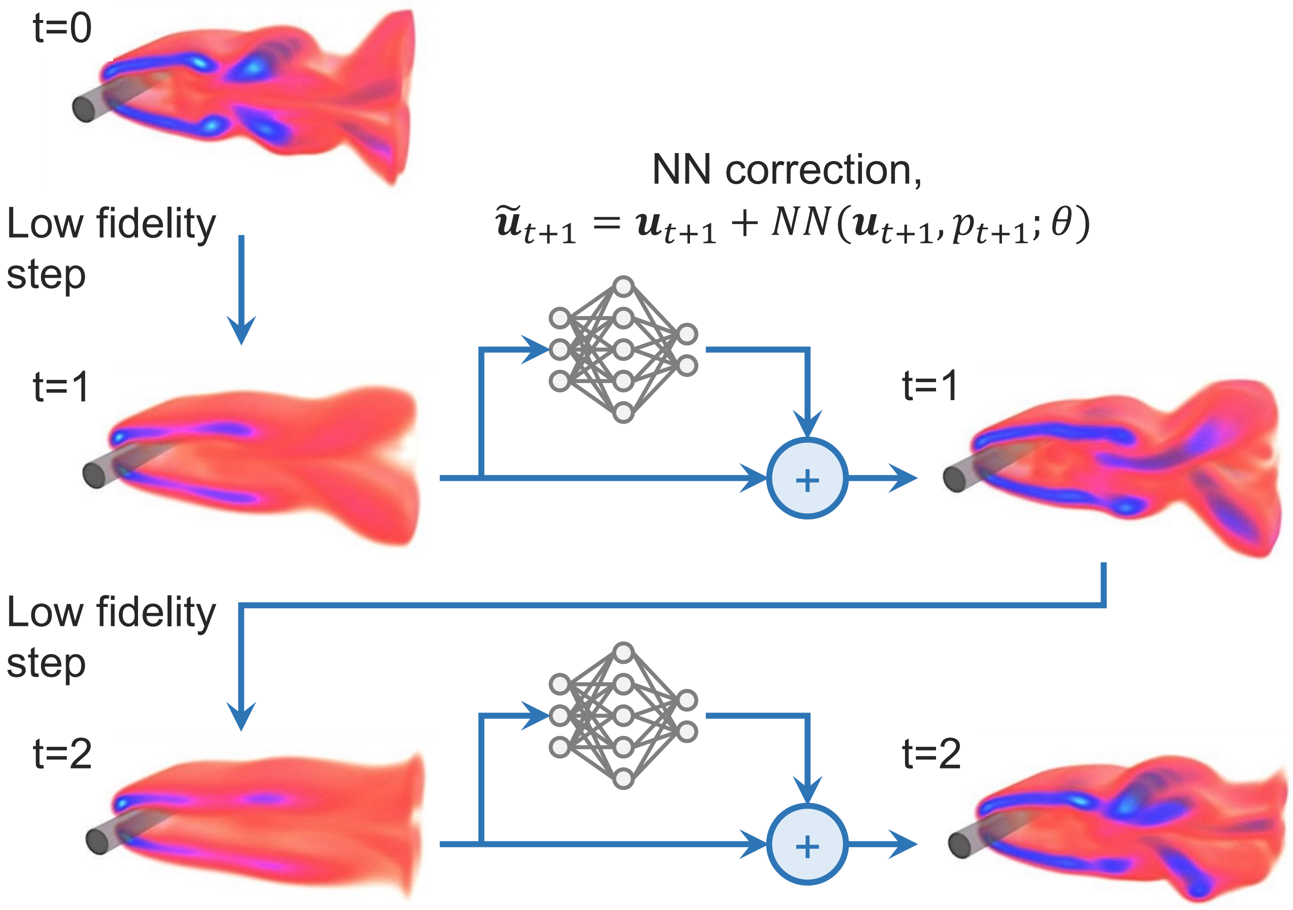
32 x 32 x 64 grid cells
~10 seconds / 100 timesteps

128 x 128 x 256 cells
~1000 seconds / 100 timesteps

32 x 32 x 64 grid cells
~15 seconds / 100 timesteps

Um et al, Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers, NeurIPS (2020)

Recap – hybrid Navier-Stokes solver

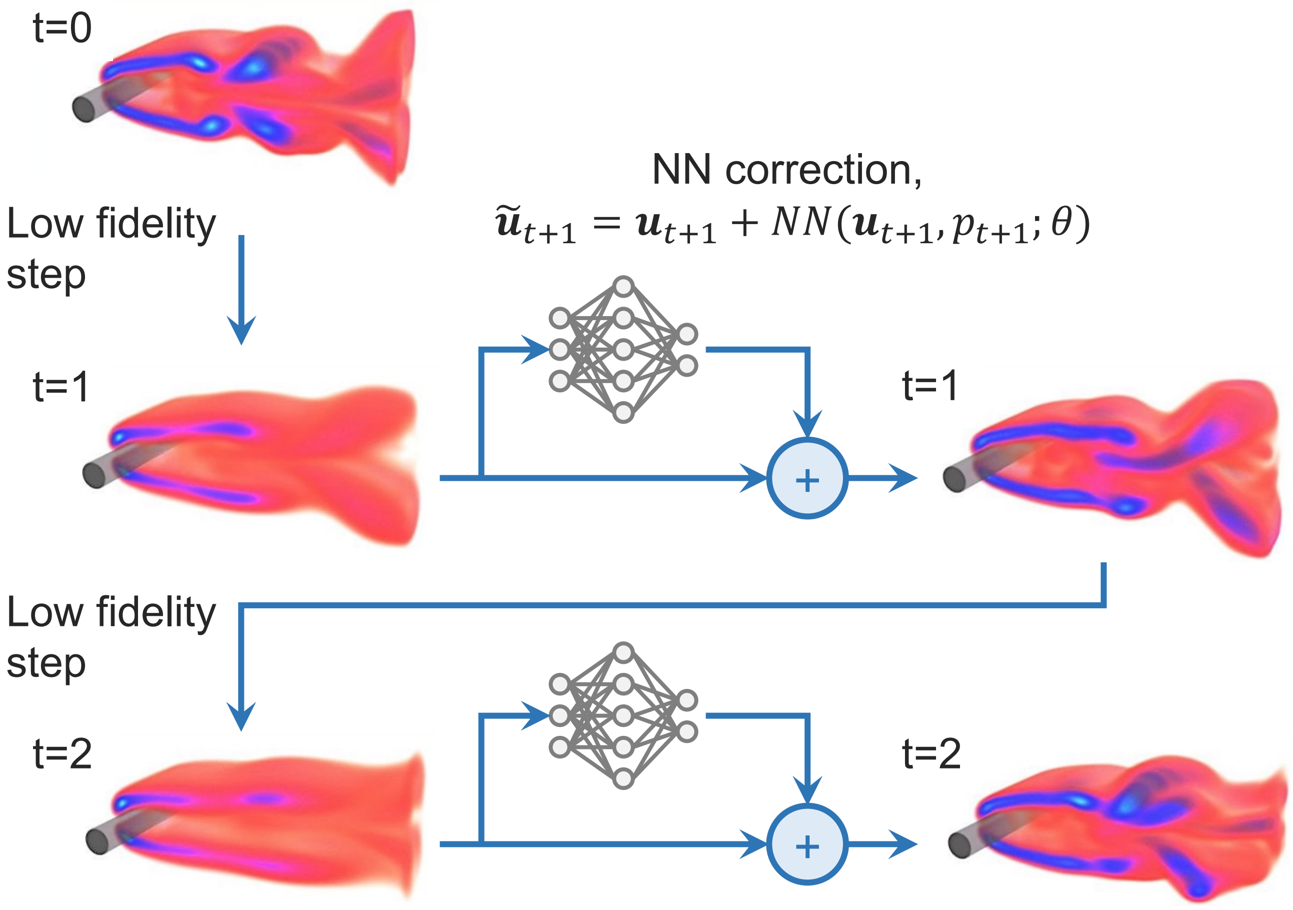


Um et al, Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers, NeurIPS (2020)

Recap – hybrid Navier-Stokes solver

```
def Hybrid_NS_solver(u_0, p_0, rho, nu, theta):
    "Pseudocode for solving NS equation, with NN correction"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

        u_t, p_t = (u_t, p_t) + NN(u_t, p_t, theta)

    return u_t, p_t

theta.requires_grad_(True)
u_T,_ = Hybrid_NS_solver(u_0, p_0, rho, nu, theta)
loss = loss_fn(u_T, u_T_true)
dtheta = torch.autograd.grad(loss, theta)
# for learning theta (training NN)
```

$$L(\theta) = \sum_{i}^{N} \sum_{t}^{T} \left\| HybridSolver_t\left(\boldsymbol{u}_{0_i}; \theta\right) - \boldsymbol{u}_t^H(\boldsymbol{u}_{0_i}) \right\|^2$$

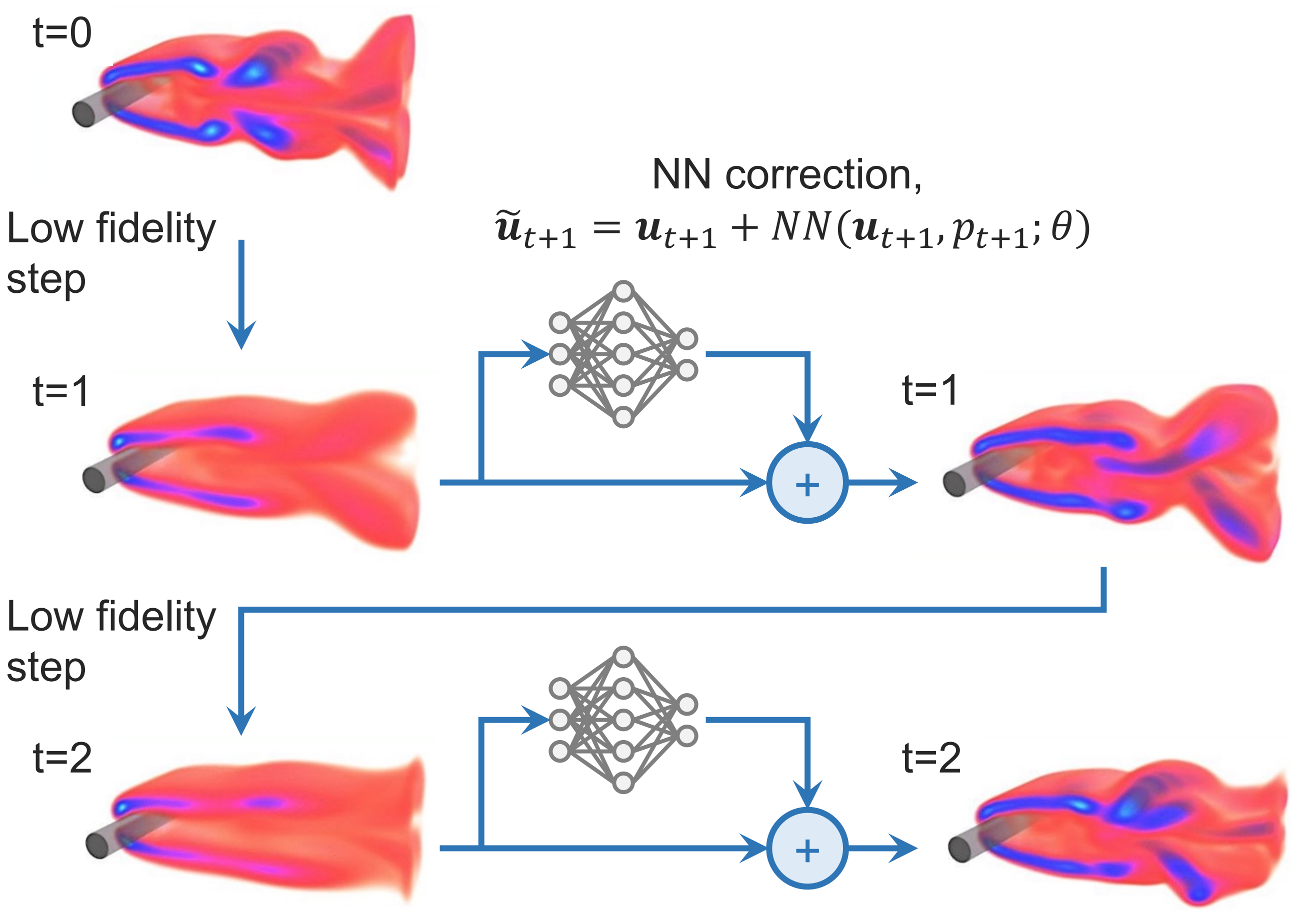


Um et al, Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers, NeurIPS (2020)

Recap – hybrid Navier-Stokes solver

```python
def Hybrid_NS_solver(u_0, p_0, rho, nu, theta):
    "Pseudocode for solving NS equation, with NN correction"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

        u_t, p_t = (u_t, p_t) + NN(u_t, p_t, theta)

    return u_t, p_t

theta.requires_grad_(True)
u_T,_ = Hybrid_NS_solver(u_0, p_0, rho, nu, theta)
loss = loss_fn(u_T, u_T_true)
dtheta = torch.autograd.grad(loss, theta)
# for learning theta (training NN)
```

$$L(\theta) = \sum_{i}^{N} \sum_{t}^{T} \left\| HybridSolver_t\left(\boldsymbol{u}_{0_i}; \theta\right) - \boldsymbol{u}_t^H\left(\boldsymbol{u}_{0_i}\right) \right\|^2$$



Key idea: **Differentiable physics** = using autodifferentiation to differentiate and learn physical algorithms

Um et al, Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers, NeurIPS (2020)

Hybrid workflows in practice

Step 1: **rewrite** your traditional scientific algorithm in an autodifferentiation framework (e.g. PyTorch/JAX)

Hybrid workflows in practice

Step 1: **rewrite** your traditional scientific algorithm in an autodifferentiation framework (e.g. PyTorch/JAX)

Step 2: make parts of this algorithm **learnable** (either to accelerate it, or to improve accuracy)

Hybrid workflows in practice

Step 1: **rewrite** your traditional scientific algorithm in an autodifferentiation framework (e.g. PyTorch/JAX)

Step 2: make parts of this algorithm **learnable** (either to accelerate it, or to improve accuracy)

Step 3: get some **training examples** of what you want the input/output of the algorithm to be

Hybrid workflows in practice

Step 1: **rewrite** your traditional scientific algorithm in an autodifferentiation framework (e.g. PyTorch/JAX)

Step 2: make parts of this algorithm **learnable** (either to accelerate it, or to improve accuracy)

Step 3: get some **training examples** of what you want the input/output of the algorithm to be

Step 4: train your algorithm by **(auto)differentiating** through it and using gradient descent

Hybrid workflows in practice

Step 1: **rewrite** your traditional scientific algorithm in an autodifferentiation framework (e.g. PyTorch/JAX)

Step 2: make parts of this algorithm **learnable** (either to accelerate it, or to improve accuracy)

Step 3: get some **training examples** of what you want the input/output of the algorithm to be

Step 4: train your algorithm by **(auto)differentiating** through it and using gradient descent

Bonus: your code now runs on the GPU!

Summary

- Hybrid approaches insert learnable components **inside** traditional algorithms
- Autodifferentiation is **the** key **enabler** for SciML
  - Allows hybrid approaches to be trained end-to-end
  - Is an incredibly general and powerful tool

Course timeline

Tutorials		Lectures			
Mon 12:15-14:00 HG E 5		*Wed 08:15-10:00 ML H 44*		*Fri 12:15-13:00 ML H 44*	
19.02.		21.02.	Course introduction	23.02.	Introduction to deep learning I
26.02.	Introduction to PyTorch	28.02.	Introduction to deep learning II	01.03.	Introduction to PDEs
04.03.	Simple DNNs in PyTorch	06.03.	Physics-informed neural networks – introduction	08.03.	Physics-informed neural networks - limitations
11.03.	Implementing PINNs I	13.03.	Physics-informed neural networks – extensions	15.03.	Physics-informed neural networks – theory I
18.03.	Implementing PINNs II	20.03.	Physics-informed neural networks – theory II	22.03.	Supervised learning for PDEs I
25.03.	Operator learning I	27.03.	Supervised learning for PDEs II	29.03.	
01.04.		03.04.		05.04.	
08.04.	Operator learning II	10.04.	Introduction to operator learning I	12.04.	Introduction to operator learning II
15.04.		17.04.	Convolutional neural operators	19.04.	Time-dependent neural operators
22.04.	GNNs	24.04.	Large-scale neural operators	26.04.	Attention as a neural operator
29.04.	Transformers	01.05.		03.05.	Windowed attention and scaling laws
06.05.	Diffusion models	08.05.	Introduction to hybrid workflows I	10.05.	**Introduction to hybrid workflows II**
13.05.	Coding autodiff from scratch	15.05.	Neural differential equations	17.05.	Introduction to JAX
20.05.		22.05.	Symbolic regression and model discovery	24.05.	Course summary
27.05.	Intro to JAX / Neural ODEs	29.05.	Guest lecture: AlphaFold	31.05.	Guest lecture: AlphaFold

Lecture overview

- Coding a simple hybrid approach in PyTorch
- Hybrid workflows for solving inverse problems

Lecture overview

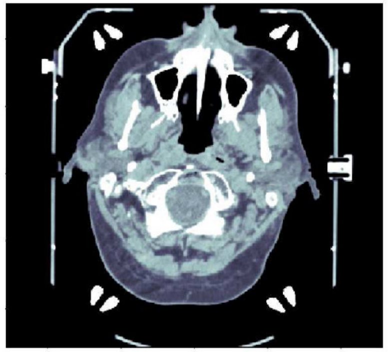
- Coding a simple hybrid approach in PyTorch
- Hybrid workflows for solving inverse problems

Learning objectives

- Be able to code a simple hybrid approach in PyTorch
- Understand more advanced hybrid workflows

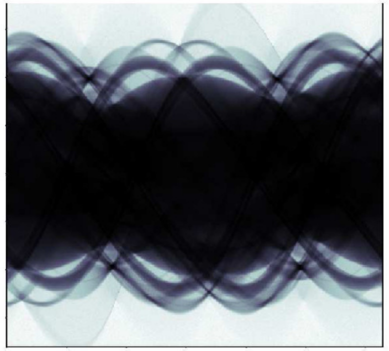
Coding a simple hybrid approach in PyTorch

Computed tomography



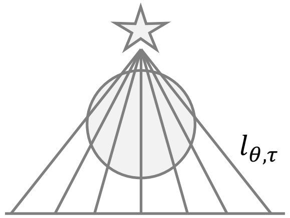
Ground truth computed tomography image

$a(x, y)$



Resulting tomographic data (sinogram)

$$b(\theta, \tau) = F(a) = I_0 e^{-\int_{l_{\theta,\tau}} a(x,y)\, ds}$$



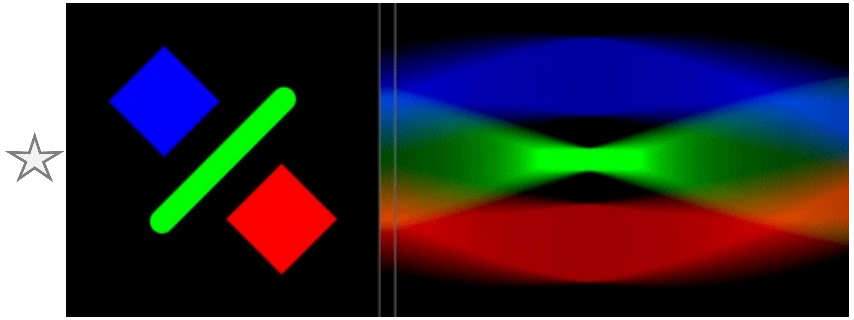
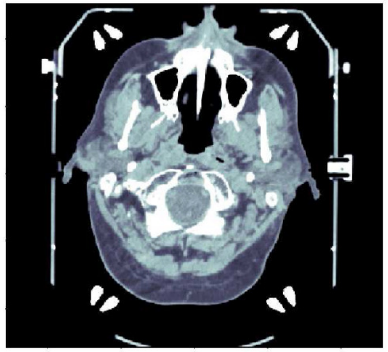


Image source: Wikipedia

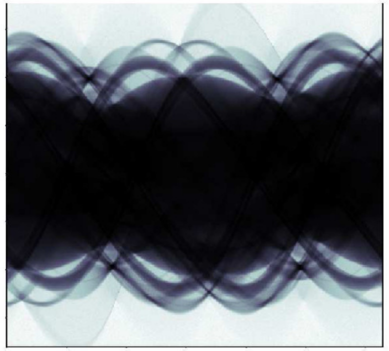
Adler et al, Solving ill-posed inverse problems using iterative deep neural networks, Inverse Problems (2017)

Computed tomography – inverse problem



Ground truth computed tomography image

$a(x, y)$



Resulting tomographic data (sinogram)

$$b(\theta, \tau) = F(a) = I_0 e^{-\int_{l_{\theta,\tau}} a(x,y)\, ds}$$

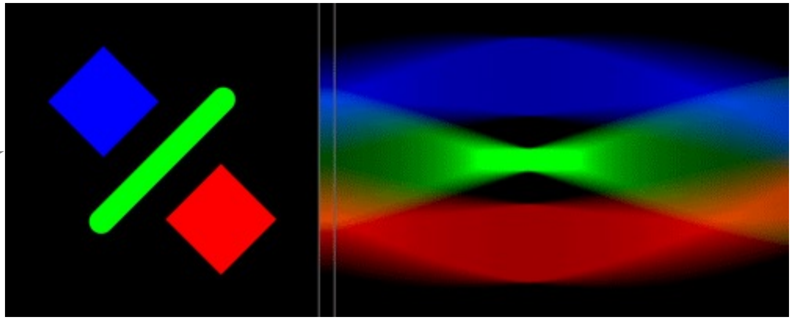
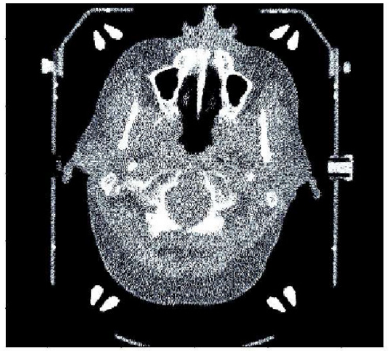
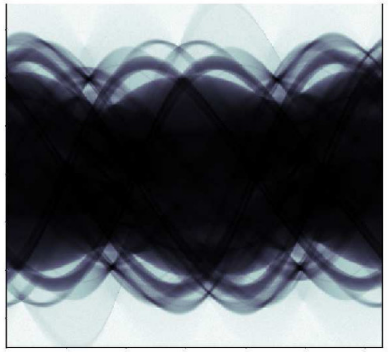


Image source: Wikipedia



Result of inverse algorithm

$\hat{a}$



Observed sinogram

b

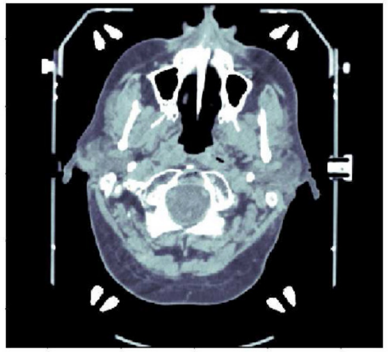
$$b = F(a)$$

a = set of input conditions

F = physical model of the system

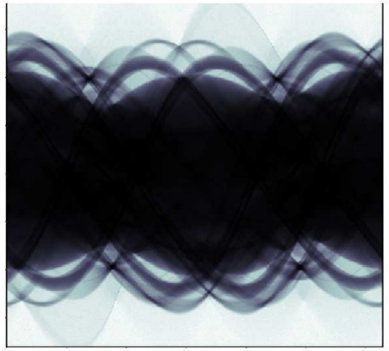
b = resulting properties given F and a

Solving the inverse problem



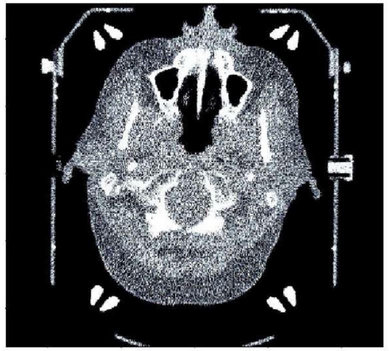
Ground truth computed tomography image

$a(x, y)$



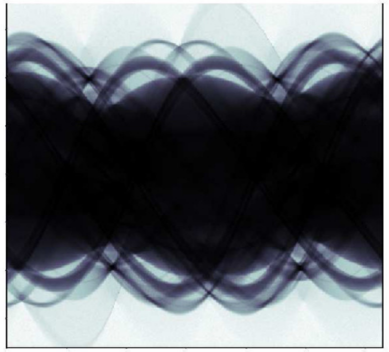
Resulting tomographic data (sinogram)

$b(\theta, \tau) = F(a) = I_0 e^{-\int_{l_{\theta,\tau}} a(x,y)\, ds}$



Result of inverse algorithm

$\hat{a}$



Observed sinogram

b

This problem can be framed as an **optimisation** problem:

$$\min_{\hat{a}} \|b - F(\hat{a})\|^2$$

Assuming F is a differentiable, we can use **gradient descent** to learn $\hat{a}$:

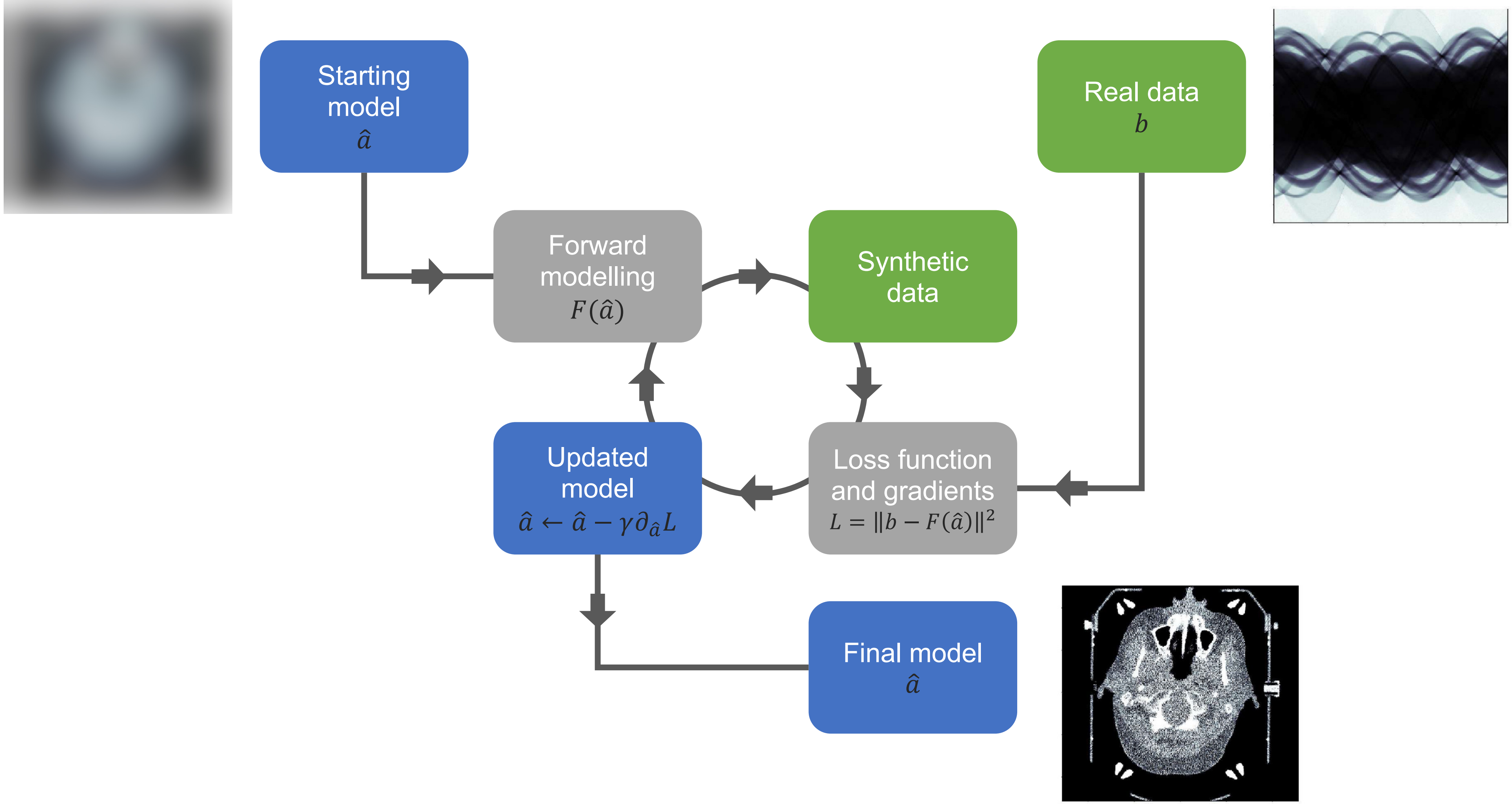
Loss function:

$$L(\hat{a}) = \|b - F(\hat{a})\|^2$$

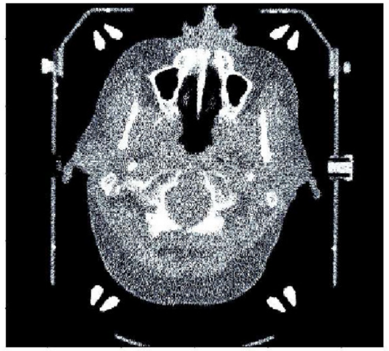
Gradient descent:

$$\hat{a} \leftarrow \hat{a} - \gamma \frac{\partial L(\hat{a})}{\partial \hat{a}}$$

Solving the inverse problem

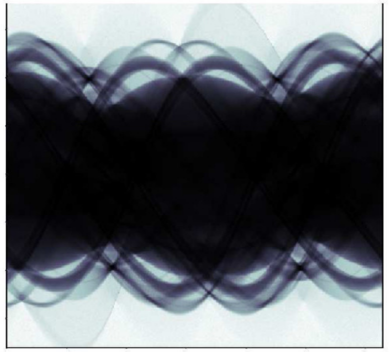


Challenges of inverse problems



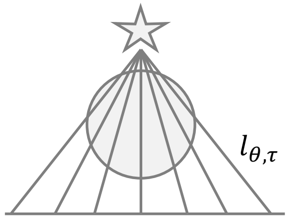
Result of inverse algorithm

$\hat{a}$



Observed sinogram

b

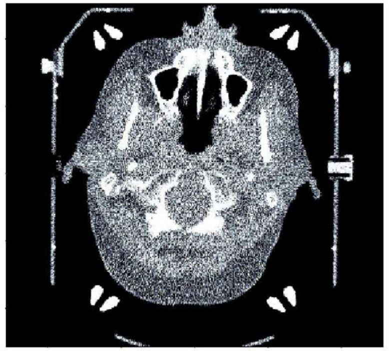


$$\min_{\hat{a}} \; \|b - F(\hat{a})\|^2$$

In general, inverse algorithms usually suffer from two major challenges:

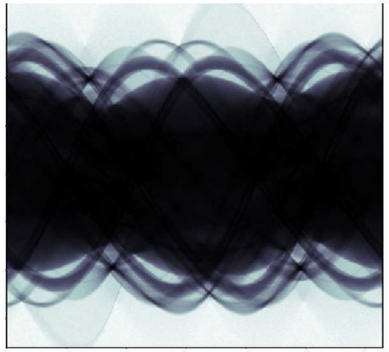
1. Poor accuracy, because they are **ill-posed** (not enough information for a unique solution):
   - Not enough measurements
   - Noisy measurements

Challenges of inverse problems



Result of inverse algorithm

$\hat{a}$



Observed sinogram

b

To improve, we need to incorporate **prior** information about the solution, for example by adding **regularization**:

$$L(\hat{a}) = ||b - F(\hat{a})||^2 + \lambda\, R(\hat{a})$$

Where, for example

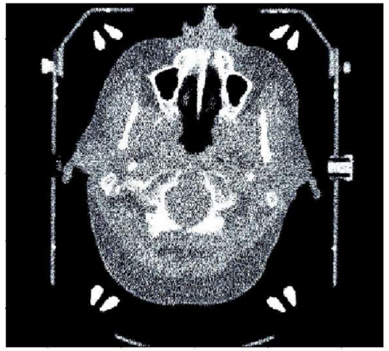
$$R(\hat{a}) = ||\nabla\hat{a}||$$

Which asserts that the output image should be "smooth" (= total variation regularization)

In general, inverse algorithms usually suffer from two major challenges:

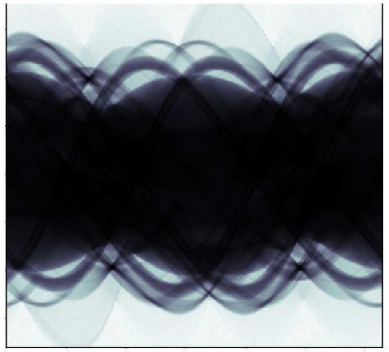
1. Poor accuracy, because they are **ill-posed** (not enough information for a unique solution):
    - Not enough measurements
    - Noisy measurements

Challenges of inverse problems



Result of inverse algorithm

$\hat{a}$



Observed sinogram

b

To improve, we need to incorporate **prior** information about the solution, for example by adding **regularization**:

$$L(\hat{a}) = ||b - F(\hat{a})||^2 + \lambda\, R(\hat{a})$$

Where, for example

$$R(\hat{a}) = ||\nabla \hat{a}||$$

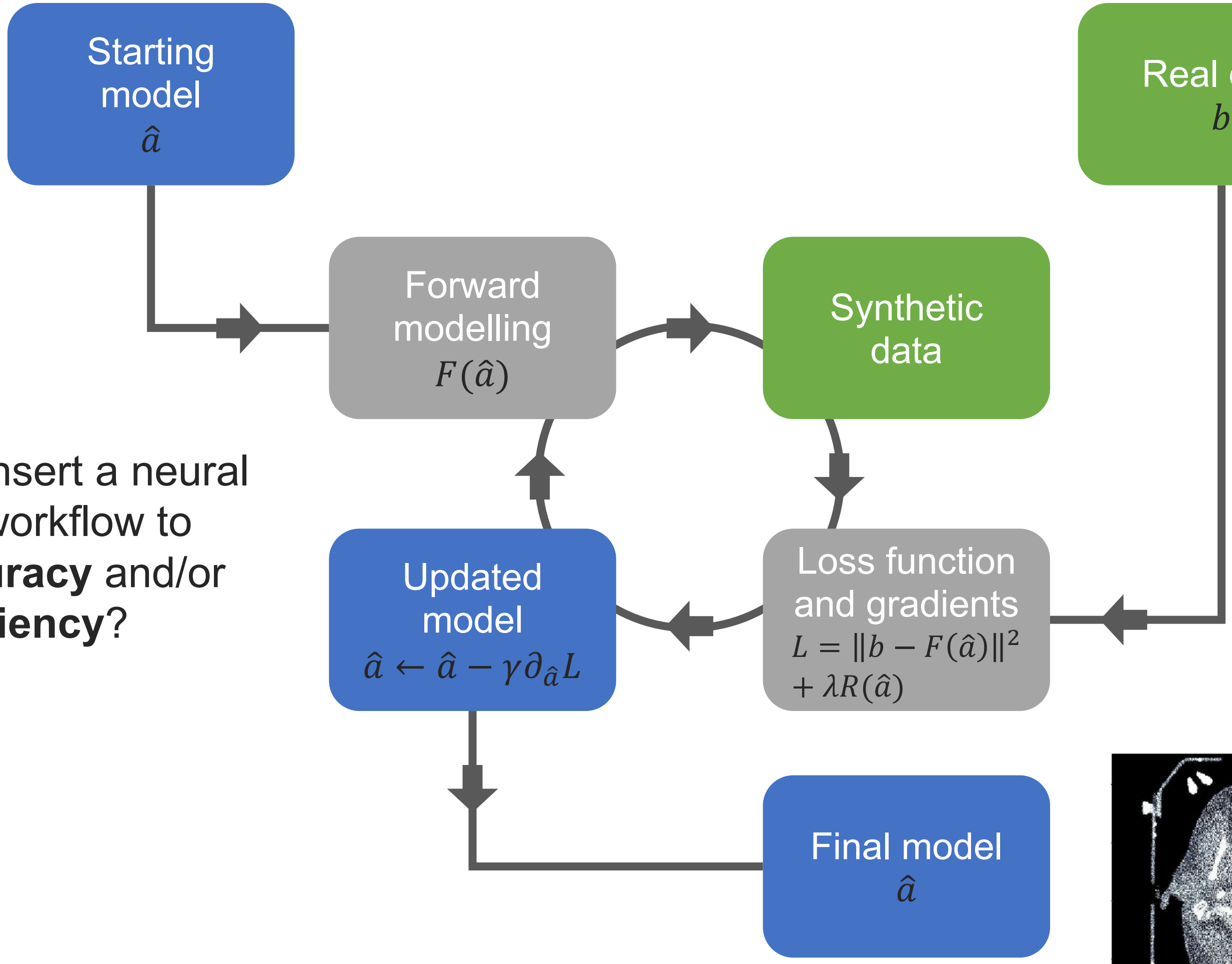
Which asserts that the output image should be "smooth" (= total variation regularization)

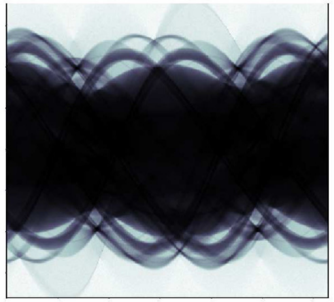
In general, inverse algorithms usually suffer from two major challenges:

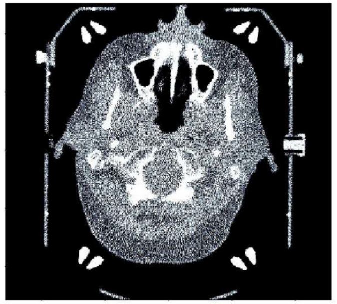
1. Poor accuracy, because they are **ill-posed** (not enough information for a unique solution):
   - Not enough measurements
   - Noisy measurements

2. Extremely computationally **expensive**, because forward modelling must be carried out thousands of times

Solving the inverse problem

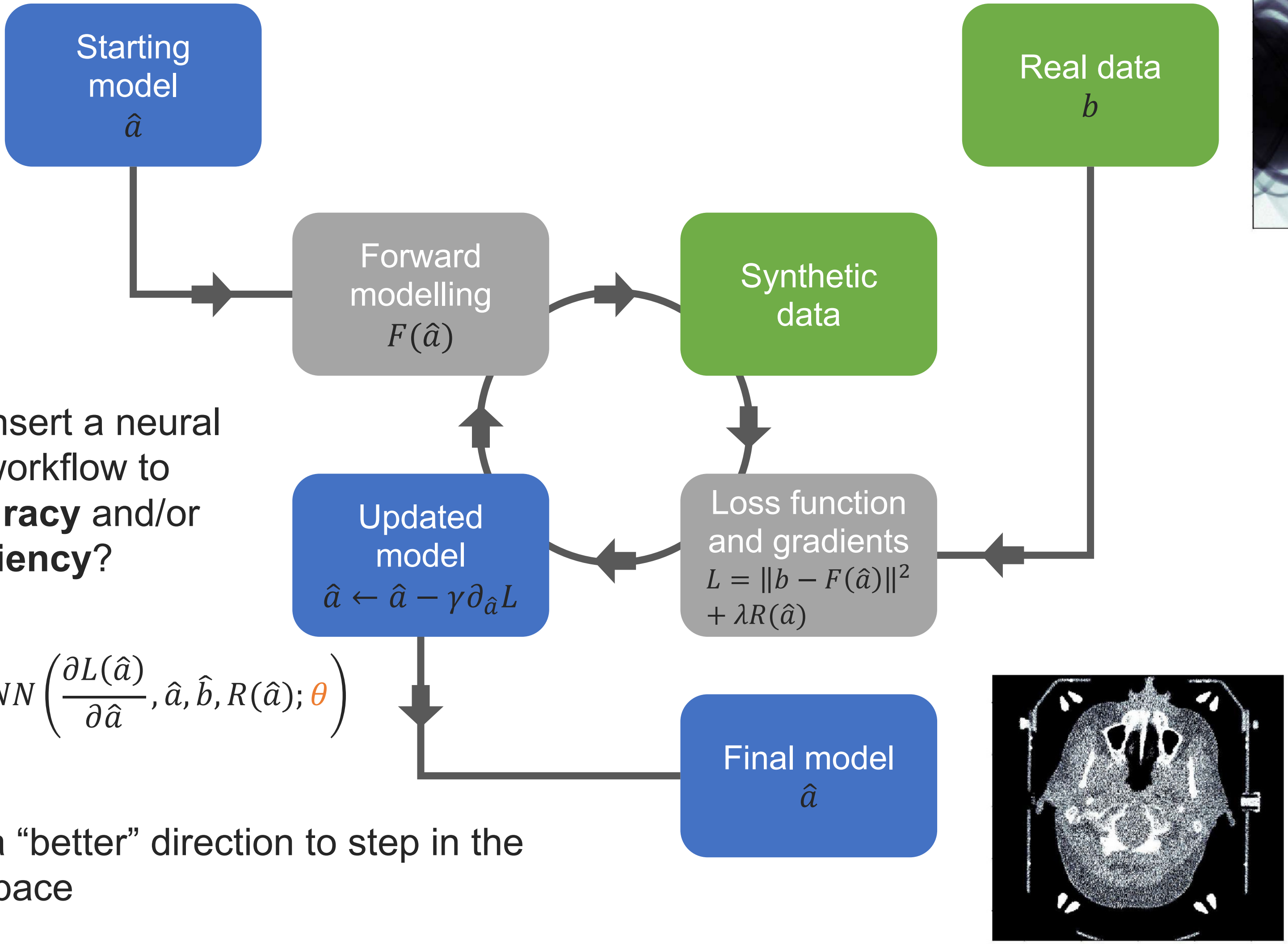


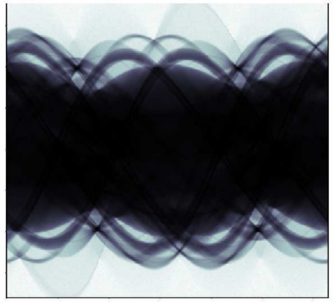




Where can we insert a neural network in this workflow to
1) improve **accuracy** and/or
2) improve **efficiency**?

Hybrid computed tomography





Where can we insert a neural network in this workflow to
1) improve **accuracy** and/or
2) improve **efficiency**?

$$\frac{\partial L(\hat{a})}{\partial \hat{a}} \leftarrow NN\left(\frac{\partial L(\hat{a})}{\partial \hat{a}}, \hat{a}, \hat{b}, R(\hat{a}); \theta\right)$$

Idea: **learn** a “better” direction to step in the parameter space

Hybrid computed tomography

```
def X_ray_tomography(a_hat_0, b):
    "Pseudocode for carrying out X ray tomography"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        a_hat -= gamma*da

    return a_hat
```

1. Start with initial guess $\hat{a}$
2. Loop:
    1. Compute gradient, $\frac{\partial L(\hat{a})}{\partial \hat{a}}$
    2. Take gradient descent step,

$$\hat{a} \leftarrow \hat{a} - \gamma \frac{\partial L(\hat{a})}{\partial \hat{a}}$$

Hybrid computed tomography

```python
def X_ray_tomography(a_hat_0, b):
    "Pseudocode for carrying out X ray tomography"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        a_hat -= gamma*da

    return a_hat
```

1. Start with initial guess $\hat{a}$
2. Loop:
    1. Compute gradient, $\frac{\partial L(\hat{a})}{\partial \hat{a}}$
    2. Take gradient descent step,

$$\hat{a} \leftarrow \hat{a} - \gamma \frac{\partial L(\hat{a})}{\partial \hat{a}}$$

```python
def Hybrid_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta)
        a_hat -= gamma*da

    return a_hat
```

1. Start with initial guess $\hat{a}$
2. Loop:
    1. Compute gradient, $\frac{\partial L(\hat{a})}{\partial \hat{a}}$
    2. Take **learned** gradient descent step,

$$\hat{a} \leftarrow \hat{a} - \gamma\, NN\left(\frac{\partial L(\hat{a})}{\partial \hat{a}}, \hat{a}, \hat{b}, R(\hat{a}); \theta\right)$$

Hybrid computed tomography

```
def X_ray_tomography(a_hat_0, b):
    "Pseudocode for carrying out X ray tomography"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        a_hat -= gamma*da

    return a_hat
```

```
def Hybrid_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta)
        a_hat -= gamma*da

    return a_hat
```

- How do we train this hybrid approach (learn θ)?

1. Start with initial guess $\hat{a}$
2. Loop:
    1. Compute gradient, $\frac{\partial L(\hat{a})}{\partial \hat{a}}$
    2. Take **learned** gradient descent step,

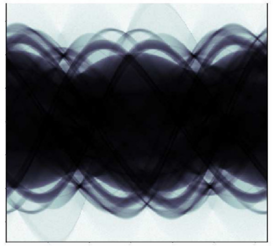
$$\hat{a} \leftarrow \hat{a} - \gamma\, NN\left(\frac{\partial L(\hat{a})}{\partial \hat{a}}, \hat{a}, \hat{b}, R(\hat{a}); \theta\right)$$

Hybrid computed tomography

Input to function:

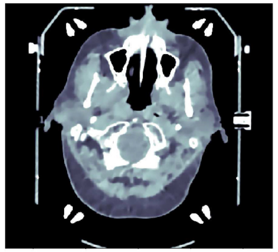
$\hat{a}_0$

b



Output:

$\hat{a}$

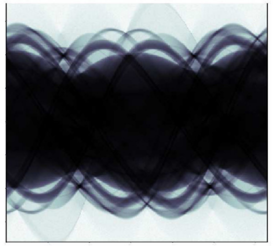


```
def Hybrid_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta)
        a_hat -= gamma*da

    return a_hat
```

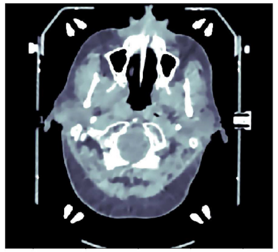
Hybrid computed tomography

Input to function:

$\hat{a}_0$ 

b 

Output:

$\hat{a}$ 

```python
def Hybrid_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta)
        a_hat -= gamma*da

    return a_hat
```

We train this hybrid approach using lots of examples of inputs $(\hat{a}_0, b)$ and outputs (a) and the loss function

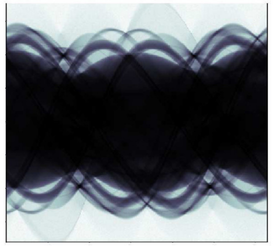
$$L(\theta) = \sum_i^N \|H(\hat{a}_{0\,i}, b_i; \theta) - a_i\|^2$$

Hybrid computed tomography

Input to function:

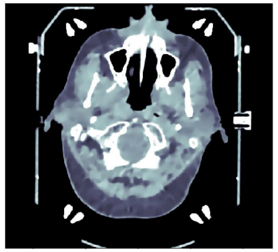
$\hat{a}_0$

b



Output:

$\hat{a}$



```
def Hybrid_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta)
        a_hat -= gamma*da

    return a_hat

# learn NN parameters
theta.requires_grad_(True)
for i in range(0, n_steps2):
    a, b = # train NN using many example inverse problems
    a_hat = Hybrid_X_ray_tomography(a_hat_0, b, theta)
    loss = loss_fn(a, a_hat)
    dtheta = torch.autograd.grad(loss, theta)
    theta -= gamma*dtheta
```

We train this hybrid approach using lots of examples of inputs $(\hat{a}_0, b)$ and outputs (a) and the loss function

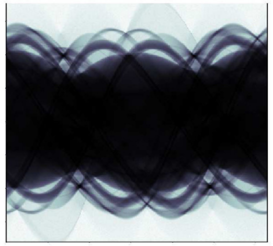
$$L(\theta) = \sum_{i}^{N} \|H(\hat{a}_{0\,i}, b_i; \theta) - a_i\|^2$$

Hybrid computed tomography

Input to function:

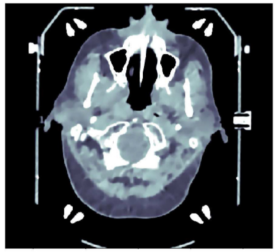
$\hat{a}_0$

b



Output:

$\hat{a}$



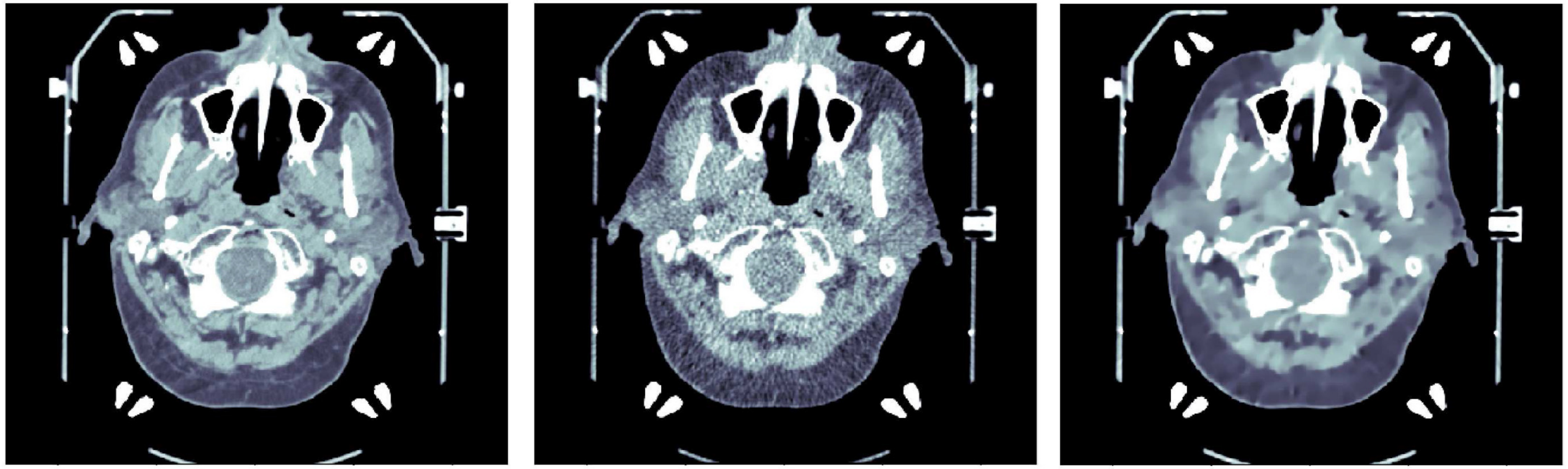
We train this hybrid approach using lots of examples of inputs $(\hat{a}_0, b)$ and outputs (a) and the loss function

$$L(\theta) = \sum_i^N \|H(\hat{a}_{0\,i}, b_i; \theta) - a_i\|^2$$

```
def Hybrid_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta)
        a_hat -= gamma*da

    return a_hat

# learn NN parameters
theta.requires_grad_(True)
for i in range(0, n_steps2):
    a, b = # train NN using many example inverse problems
    a_hat = Hybrid_X_ray_tomography(a_hat_0, b, theta)
    loss = loss_fn(a, a_hat)
    dtheta = torch.autograd.grad(loss, theta)
    theta -= gamma*dtheta
```

“Gradient descent on gradient descent”
“Learned gradient descent”
“Learning to learn”

Hybrid computed tomography



Ground truth | Traditional inversion | **Learned gradient descent**

Adler et al, Solving ill-posed inverse problems using
iterative deep neural networks, Inverse Problems (2017)

Adding even more flexibility

- We can use **more** than one learnable component if we want!
- Where else would it be useful to add another?

```python
def Hybrid_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta)
        a_hat -= gamma*da

    return a_hat

# learn NN parameters
theta.requires_grad_(True)
for i in range(0, n_steps2):
    a, b = # train NN using many example inverse problems
    a_hat = Hybrid_X_ray_tomography(a_hat_0, b, theta)
    loss = loss_fn(a, a_hat)
    dtheta = torch.autograd.grad(loss, theta)
    theta -= gamma*dtheta
```

Adding even more flexibility

```python
def Hybrid2_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0

    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + theta[0]*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta[1])
        a_hat -= gamma*da

    return a_hat

# learn NN parameters
theta.requires_grad_(True)
for i in range(0, n_steps2):
    a, b = # train NN using many example inverse problems
    a_hat = Hybrid2_X_ray_tomography(a_hat_0, b, theta)
    loss = loss_fn(a, a_hat)
    dtheta = torch.autograd.grad(loss, theta)
    theta -= gamma*dtheta
```

Idea 2: learn regularisation **hyperparameter** too

```python
def Hybrid_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta)
        a_hat -= gamma*da

    return a_hat

# learn NN parameters
theta.requires_grad_(True)
for i in range(0, n_steps2):
    a, b = # train NN using many example inverse problems
    a_hat = Hybrid_X_ray_tomography(a_hat_0, b, theta)
    loss = loss_fn(a, a_hat)
    dtheta = torch.autograd.grad(loss, theta)
    theta -= gamma*dtheta
```

Idea 1: **learn** a “better” direction to step in the parameter space

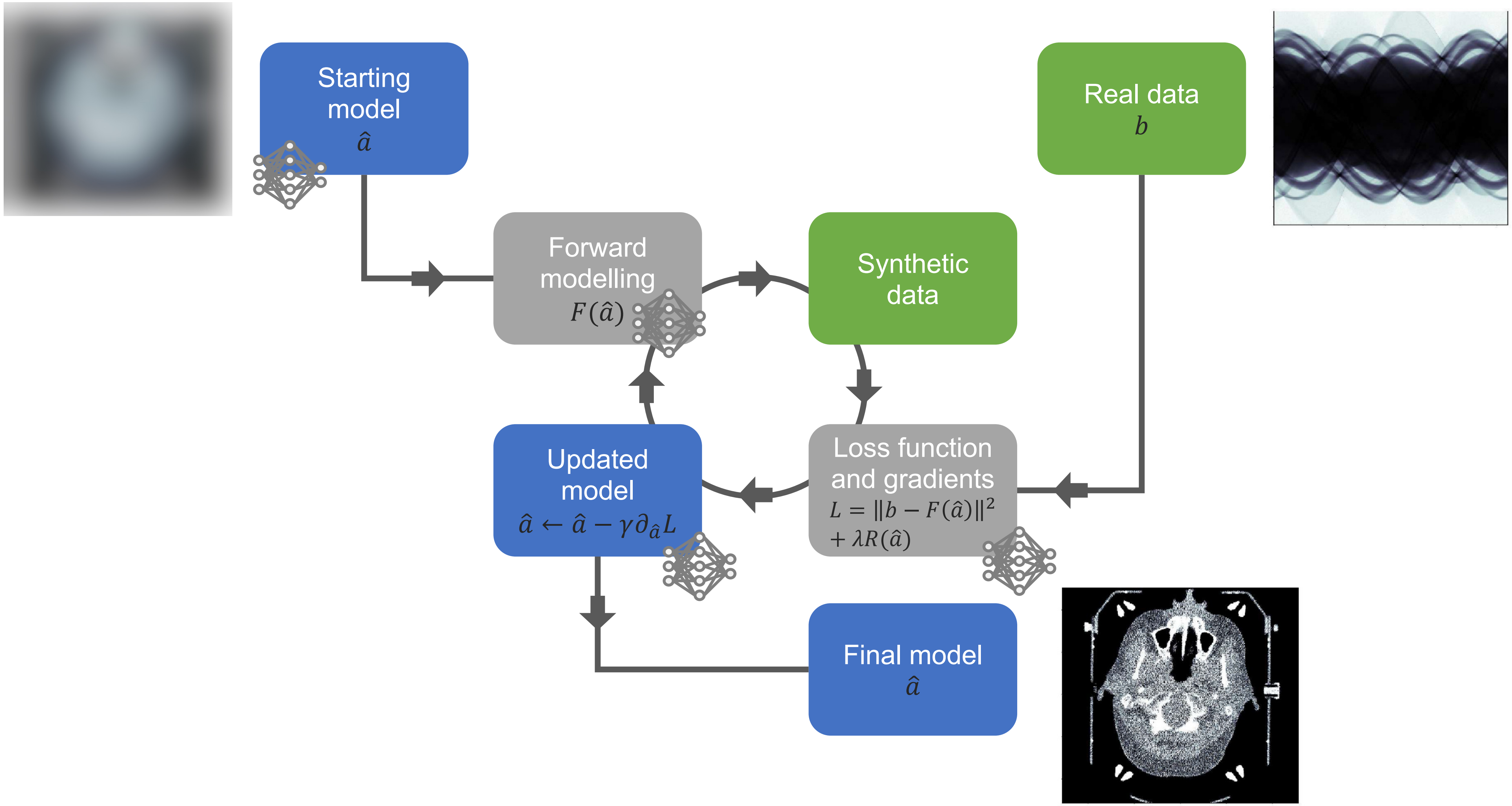
Adding even more flexibility

```python
def Hybrid2_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0

    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + theta[0]*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta[1])
        a_hat -= gamma*da

    return a_hat

# learn NN parameters
theta.requires_grad_(True)
for i in range(0, n_steps2):
    a, b = # train NN using many example inverse problems
    a_hat = Hybrid2_X_ray_tomography(a_hat_0, b, theta)
    loss = loss_fn(a, a_hat)
    dtheta = torch.autograd.grad(loss, theta)
    theta -= gamma*dtheta
```

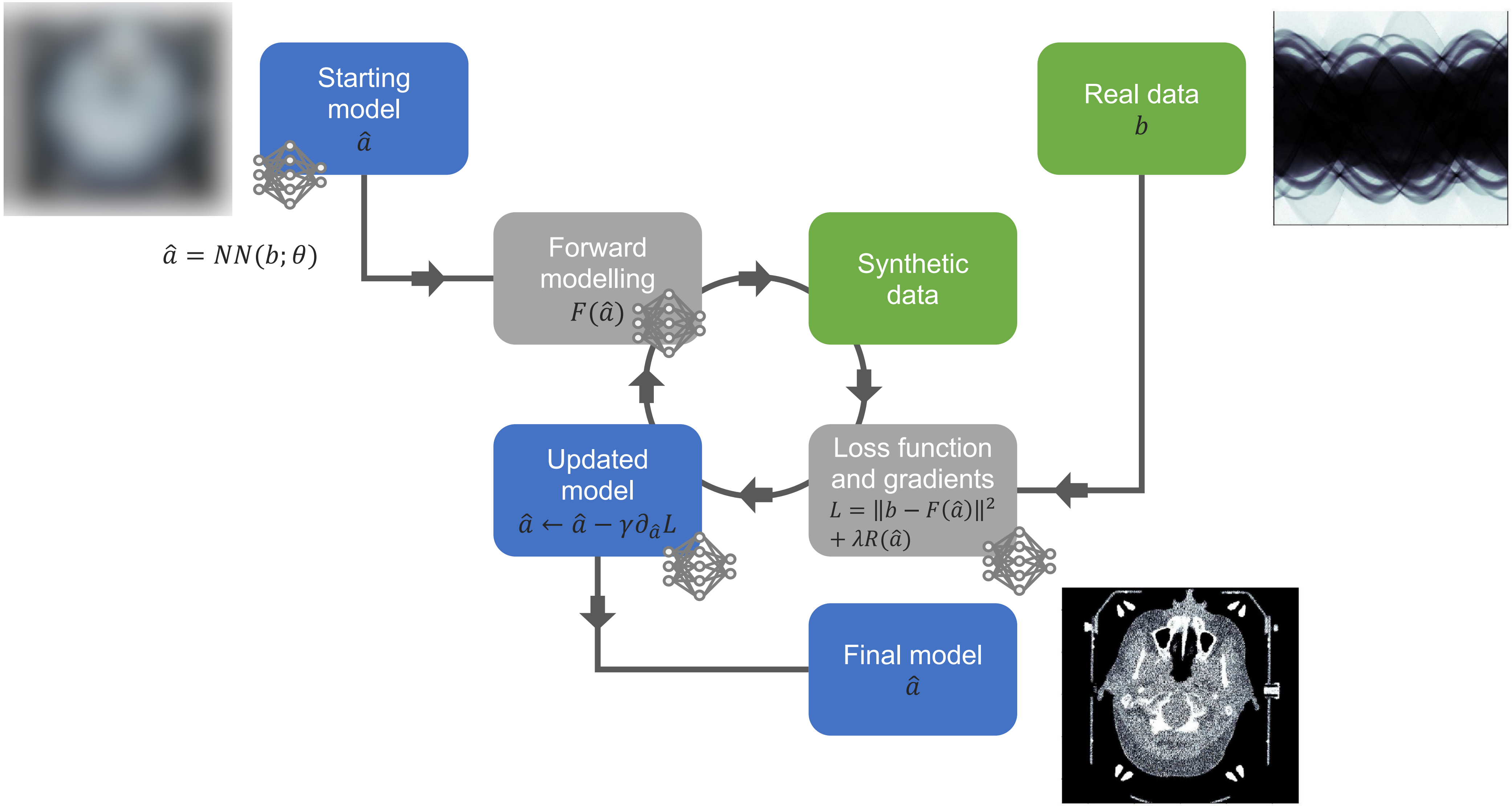
Key idea:
Traditional algorithms can be made as **learnable** (flexible) or as **unlearnable** (rigid) as you like

This allows you to balance the pros/cons of using NNs!

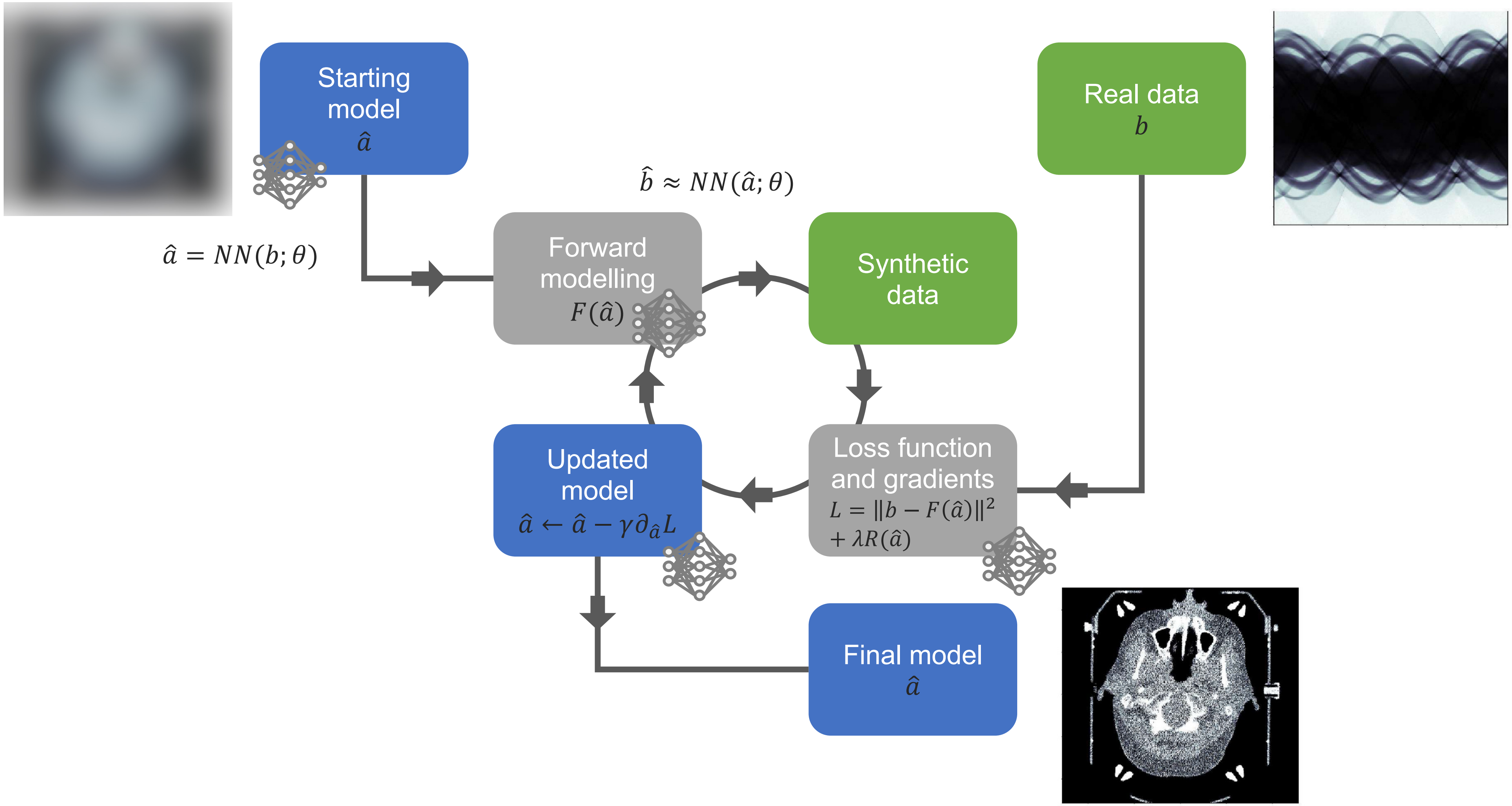
We can add learnable components everywhere!



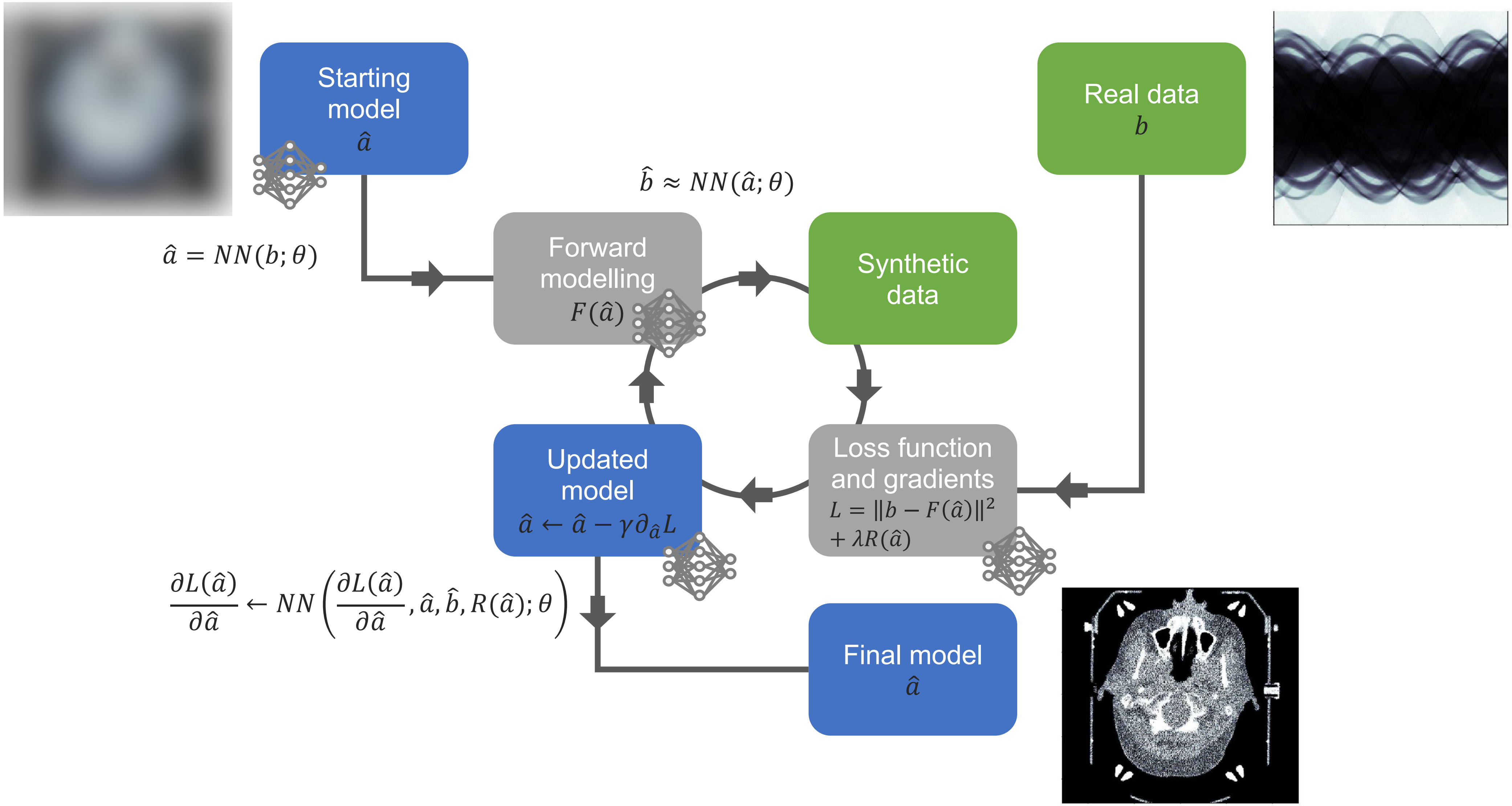
We can add learnable components everywhere!



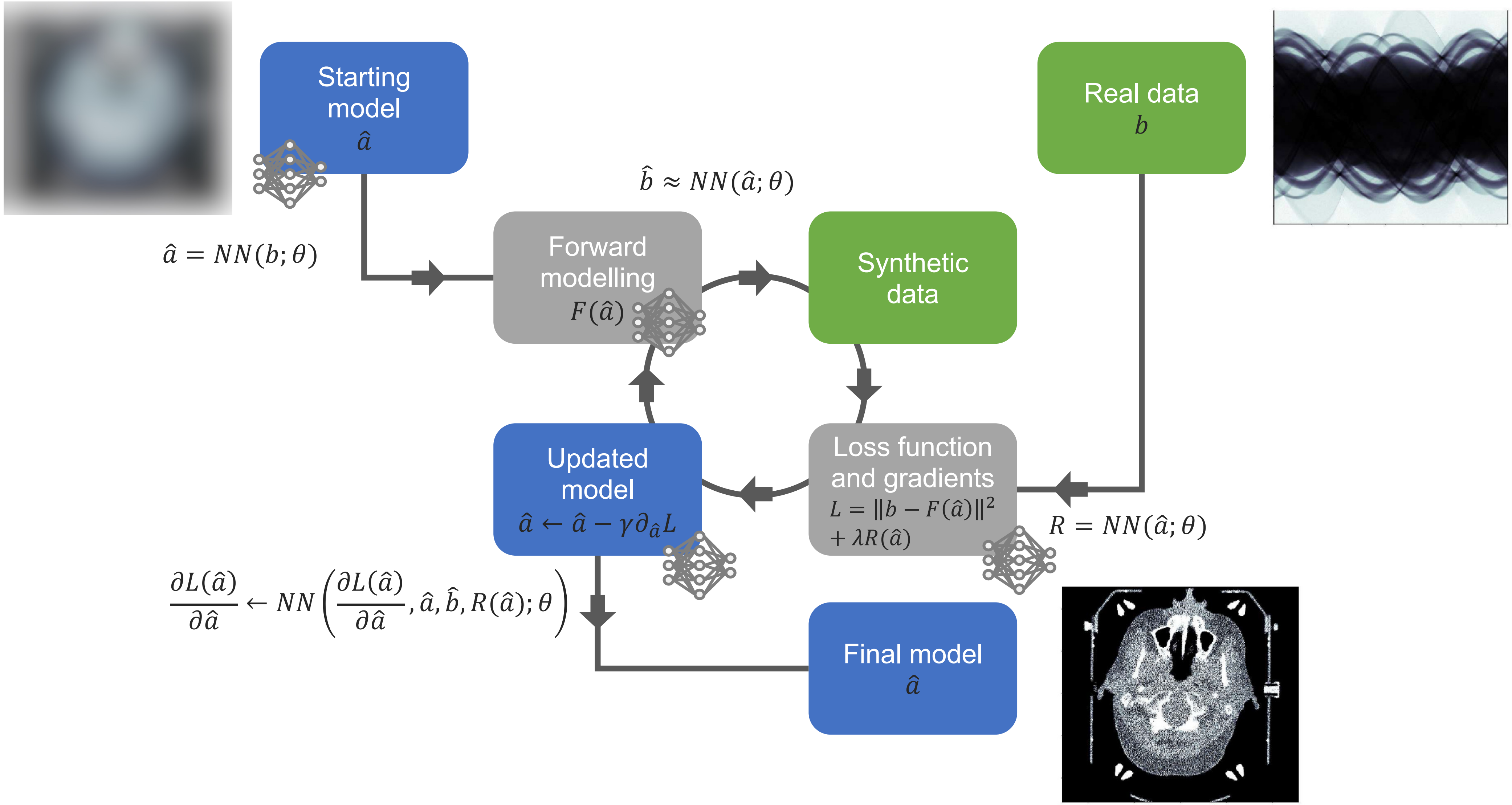
We can add learnable components everywhere!



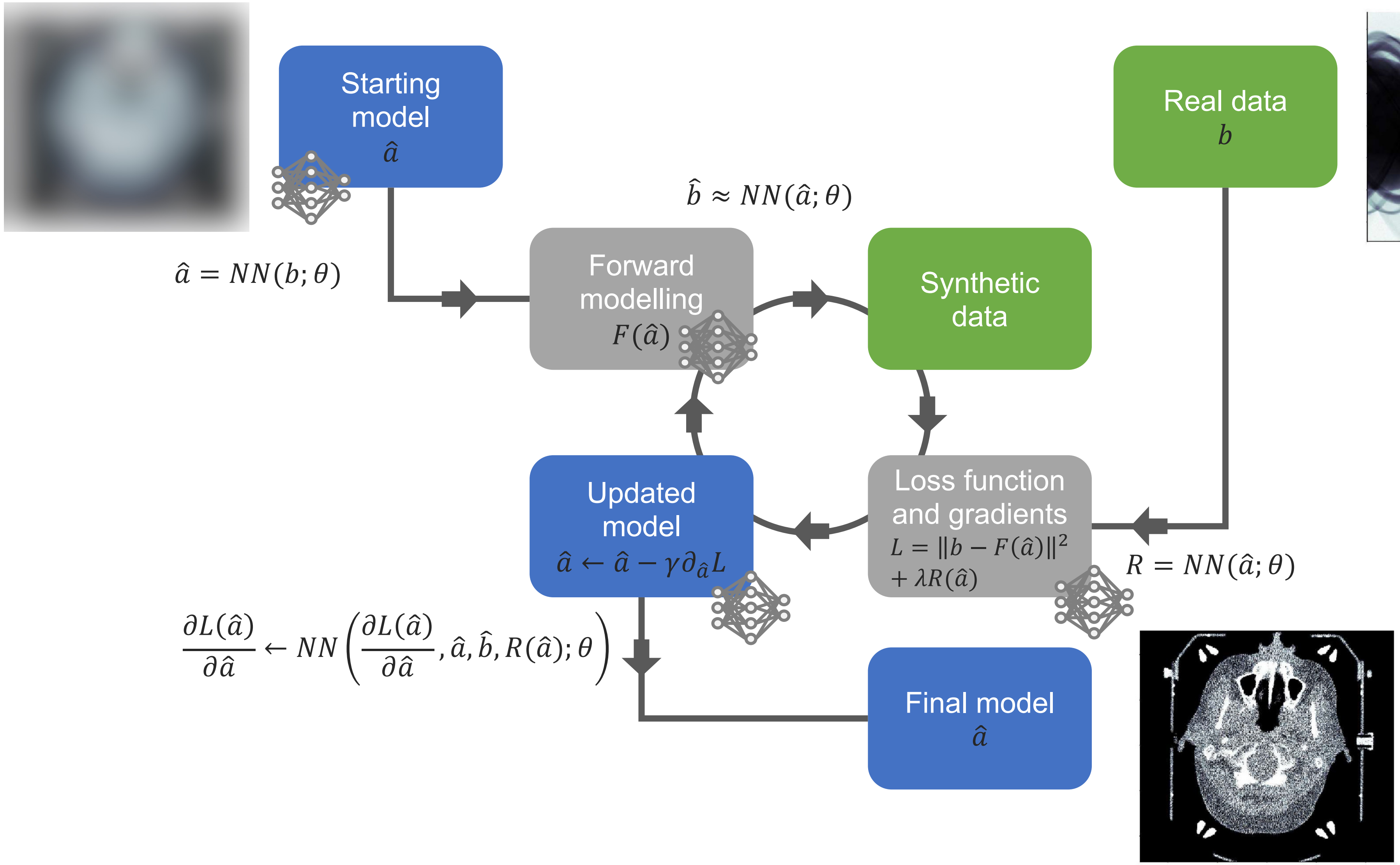
We can add learnable components everywhere!



We can add learnable components everywhere!



We can add learnable components everywhere!



$$L(\theta) = \sum_{i}^{N} \|H(\hat{a}_{0\,i}, b_i; \theta) - a_i\|^2$$

Lecture summary

- Traditional algorithms can be made as **learnable** (flexible) or as **unlearnable** (rigid) as you like

- Inside hybrid inverse algorithms, neural networks can be very effective at **learning priors** and improving **efficiency**